CS10720 Problems and Solutions

Thomas Jansen

Today: Search in a Sorted Array: Linear and Binary Search

February 15^{th}

Organisational Issues

Important ‘additional’ lecture this Friday, 4-5pm, EL 0.26
(replacing the Monday lecture that fell victim to the storm)

Important portfolio this week same contents as always
lecture summary Monday and Thursday
answer to practicals questions
(not lecture summary Friday)
same deadline as always, Friday, 7pm

Remember portfolio feedback for last week later
(I’m working on it. It’ll take more time.)

Plans for Today

Our Problem for Today

General formulation ‘Find something you are looking for.’

Problem Have a large number of different items
each with a unique identifier (called key)
in some arrangement
that is sorted in ascending order according to keys.
You are given a key, you have to find the item.

Observation used to be frequent problem for people
and still is frequent problem for computers

Plan for today learn how to solve
one specific instance of this problem on a computer
in several different ways

Problem Definition

We simplify the problem by

1. removing the items and only considering the keys
2. having integers as keys

Remark simplifies writing down implementations of solutions
but makes the problem almost pointless
however it should be clear how to use different keys
and add back items

Remember we concentrate on the essence of the search problem
without being distracted by implementation details

Input array `keys` of type `int` of length n
specific `key` of type `int`

Output if $(\exists i \in \{0, 1, \ldots, n-1\} \colon \texttt{keys}[i] = \texttt{key})$ then i, else -1

```
#define n 100000 /* just an example size */
int keys[n]; /* the array of keys */
int key; /* the key we are looking for */
int search(int *keys, int size, int key);
```

Towards a Solution

What do we want from a solution to this problem?

1. It should be correct.
2. It should be efficient.

A few stupid suggestions

- Pick a random position i, if `keys[i]==key` then return i else return -1
  Observation really fast but with high probability not correct (actually correct if `key` is not in `keys`; but otherwise correct only with probability $1/n$, thus not correct with probability $1 - 1/n$)
- Repeat n times: pick a random position i, if `keys[i]==key` then return i. Return -1 if key not found.
  Observation up to n times slower and still not correct (if key present not correct with probability $(1 - 1/n)^n \approx 1/e > 0.36$)
- Check all positions in some order. Return position where `key` found or -1 if not found. Observation correct

Implementing Our First Correct Idea

```
int search(int *keys, int size, int key) {
  int i; /* index variable for a loop over keys */

  for ( i=0; i<size; i++ ) {
    if ( keys[i] == key )
      return i;
  }
  return -1;
}
```

Does this have to be C? Can't you explain this in English?

Remark most algorithms (including this one)
in three versions in the lecture notes:
English, pseudo-code, C code
Please, read the lecture notes if something is unclear!

http://onlineted.com

Reconsidering Our First Correct Idea

```
int search(int *keys, int size, int key) {
  int i; /* index variable for a loop over keys */

  for ( i=0; i<size; i++ ) {
    if ( keys[i] == key )
      return i;
  }
  return -1;
}
```

Observation algorithm is stupid!

Observation once `keys[i] > key` we can stop looking because the keys only get bigger the further we get

Consequence stop as soon as possible, i. e. when `keys[i] > key`

Implementing Our First Correct Idea a Bit Better

```
int search(int *keys, int size, int key) {
  int i; /* index variable for a loop over keys */

  i=0; /* start searching at the beginning */
  while ( (i<size) && (keys[i]<key) ) { /* search for key */
    i++; /* move to next item */
  }
  if ( (i==size) || (keys[i]>key) ) /* 2 cases for key not found */
    return -1;
  else
    return i;
}
```

Observation correct ✓

Alternative Implementation

```
int search(int *keys, int size, int key) {
  int i; /* index variable for a loop over keys */

  i=size-1; /* start searching at the end */
  while ( (i>=0) && (keys[i]>key) ) { /* search for key */
    i--; /* move to next item */
  }
  if ( (i==-1) || (keys[i]<key) ) /* 2 cases for key not found */
    return -1;
  else
    return i;
}
```

Observation also correct ✓

Comparing Two Implementations

```
int search(int *keys, int size, int key) {
  int i=0;
  while ( (i<size) && (keys[i]<key) )
    i++;
  if ( (i==size) || (keys[i]>key) )
    return -1;
  else
    return i;
}
```

```
int search(int *keys, int size, int key) {
  int i=size-1;
  while ( (i>=0) && (keys[i]>key) )
    i--;
  if ( (i==-1) || (keys[i]<key) )
    return -1;
  else
    return i;
}
```

Observation both correct ✓ Which one is more efficient?

Observations

- In the best case both find the key in the first step
- In the worst case both search whole array and return -1
- In the average case key position (present or not present) is at each position with equal probability
  both look at $\frac{1}{n} \cdot 1 + \frac{1}{n} \cdot 2 + \cdots + \frac{1}{n} \cdot n = \frac{n(n+1)}{2n} = \frac{n+1}{2}$ keys ($n =$ size)
- in all cases both perform equal

Comparing Two Implementations

```
int search(int *keys, int size, int key) {
  int i=0;
  while ( (i<size) && (keys[i]<key) )
    i++;
  if ( (i==size) || (keys[i]>key) )
    return -1;
  else
    return i;
}
```

```
int search(int *keys, int size, int key) {
  int i=size-1;
  while ( (i>=0) && (keys[i]>key) )
    i--;
  if ( (i==-1) || (keys[i]<key) )
    return -1;
  else
    return i;
}
```

Observations

- both correct
- In the best case considering 1 key
  very unlikely and not very important
- in the worst case both considering n keys ($n =$ size)
- in the average case both considering $(n + 1)/2$ keys

Can we search significantly faster than this?

Improving Over Linear Search

```
int search(int *keys, int size, int key) {
  int i=0;
  while ( (i<size) && (keys[i]<key) )
    i++;
  if ( (i==size) || (keys[i]>key) )
    return -1;
  else
    return i;
}
```

```
int search(int *keys, int size, int key) {
  int i=size-1;
  while ( (i>=0) && (keys[i]>key) )
    i--;
  if ( (i==-1) || (keys[i]<key) )
    return -1;
  else
    return i;
}
```

Do you search in a phonebook
by reading it from front to back
or by reading it from back to front?

Observation we can exploit the order of the keys
much better than this
by trying to locate the relevant area more quickly

Binary Search

Idea try your luck in the middle and continue in the correct half using the same idea in that half and so on...

```
int search(int *keys, int size, int key) {
  int left, right; /* defining boundaries of search area */
  int middle; /* the ‘middle’ where we hope to find the key */
  left = 0; /* leftmost key */
  right = size-1; /* rightmost key */
  while ( left <= right ) {
    middle = left + (right-left)/2; /* midpoint */
    if ( keys[middle] == key )
      return middle;
    if ( keys[middle] < key ) /* need to look in right half */
      left = middle+1; /* can exclude middle */
    else /* need to look in left half */
      right = middle-1; /* can exclude middle */
  }
  return -1;
}
```

Binary Search: Correctness

```
while ( left <= right ) {
  middle = left + (right-left)/2; /* midpoint */
  if ( keys[middle] == key )
    return middle;
  if ( keys[middle] < key ) /* need to look in right half */
    left = middle+1; /* can exclude middle */
  else /* need to look in left half */
    right = middle-1; /* can exclude middle */
}
return -1;
```

Is this correct? yes (but only ‘hand waving’ now; more in March)

- if key not present
  - code never returns anything but -1
  - $(\text{right} - \text{left})$ gets smaller each round
  - when $(\text{right} - \text{left}) < 0$ code returns -1 ✔
- if key present
  - initially key is between left and right (inclusive)
  - this is always the case
  - at the latest when $(\text{right} - \text{left}) = 0$

  key is found and code returns correct position ✔

Binary Search: Efficiency

```
int search(int *keys, int size, int key) {
  int left, right; /* defining boundaries of search area */
  int middle; /* the ‘middle’ where we hope to find the key */
  left = 0; /* leftmost key */
  right = size-1; /* rightmost key */
  while ( left <= right ) {
    middle = left + (right-left)/2; /* midpoint */
    if ( keys[middle] == key )
      return middle;
    if ( keys[middle] < key ) /* need to look in right half */
      left = middle+1; /* can exclude middle */
    else /* need to look in left half */
      right = middle-1; /* can exclude middle */
  }
  return -1;
}
```

Is this more efficient than linear search?

Observation   in the best case considering 1 key
(the same as for both variants of linear search)
but equally unlikely

Let's   only consider the worst case
(since the average case is a bit more difficult)
i. e., case where key is not present

Binary Search: Efficiency in the Worst Case

```
int search(int *keys, int size, int key) {
  int left, right; /* defining boundaries of search area */
  int middle; /* the ‘middle’ where we hope to find the key */
  left = 0; /* leftmost key */
  right = size-1; /* rightmost key */
  while ( left <= right ) {
    middle = left + (right-left)/2; /* midpoint */
    if ( keys[middle] == key )
      return middle;
    if ( keys[middle] < key ) /* need to look in right half */
      left = middle+1; /* can exclude middle */
    else /* need to look in left half */
      right = middle-1; /* can exclude middle */
  }
  return -1;
}
```

How many keys are considered when the key is not present?

- in each round 2 keys (the same key twice)
- number of keys is twice the number of rounds
- What is the number of rounds in the worst case?
- Remember either left changes to middle $+1$ or right changes to middle -1 with middle $=$ left $+$ (right $-$ left)$/2$

Binary Search: Number of Rounds in the Worst Case

Remember either left changes to middle $+ 1$
or right changes to middle $- 1$
with middle $=$ left $+$ (right $-$ left)$/2$

In the following $l =$ left$, r =$ right$, n =$ size

Case 1 range changes from $r - l$ to $r - \left(l + \left\lfloor \frac{r-l}{2} \right\rfloor + 1\right)$

$$= (r - l) - \left(\left\lfloor \frac{r-l}{2} \right\rfloor + 1\right) = (r - l) - \left\lfloor \frac{r-l}{2} \right\rfloor - 1$$
$$\leq (r - l) - \frac{r-l}{2} = \frac{r-l}{2}$$

Case 2 range changes from $r - l$ to $\left(l + \left\lfloor \frac{r-l}{2} \right\rfloor - 1\right) - l$

$$= \left\lfloor \frac{r-l}{2} \right\rfloor - 1 < \frac{r-l}{2}$$

Remember in each round range shrinks from $r - l$ to $\leq (r - l)/2$
and round with $r - l < 1$ is last (in the worst case)

Computing Number of Rounds in the Worst Case

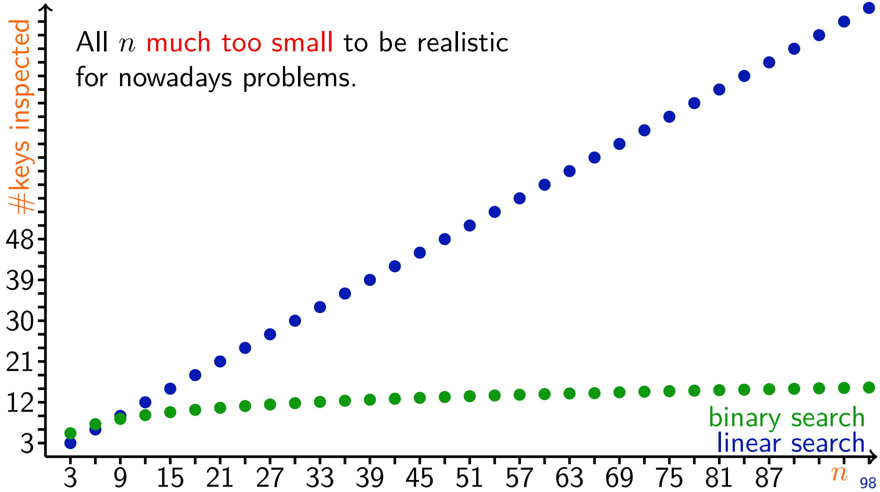
Remember in each round range shrinks from $r - l$ to $\leq (r - l)/2$
and round with $r - l < 1$ is last (in the worst case)
initially $r - l = n - 1$ (Assuming $r - l = n$ makes it worse)

Consider development of range $r - l$ (in the worst case)
$n \rightsquigarrow \frac{n}{2} \rightsquigarrow \frac{n}{4} \rightsquigarrow \frac{n}{8} \rightsquigarrow \frac{n}{16} \rightsquigarrow \cdots \rightsquigarrow X$
$n \rightsquigarrow \frac{n}{2^1} \rightsquigarrow \frac{n}{2^2} \rightsquigarrow \frac{n}{2^3} \rightsquigarrow \frac{n}{2^4} \rightsquigarrow \cdots \rightsquigarrow \frac{n}{2^k}$
with $X < 1$ with $2^k > n$
so $k = 1 + \log_2 n$ rounds are enough in the worst case
total number of rounds $\leq 1 + \log_2 n$ in the worst case

Summary binary search considers at most $2\log_2(n) + 2$ keys
while linear search considers up to n keys

Comparing Linear and Binary Search

Summary binary search considers at most $2\log_2(n) + 2$ keys while linear search considers up to n keys



Comparing Linear and Binary Search for Larger Arrays

Summary binary search considers at most $2\log_2(n) + 2$ keys while linear search considers up to n keys

Comparison for larger n

$\boldsymbol{n}$	**linear**	**binary**
10	10	< 9
100	100	< 16
1 000	1 000	< 22
10 000	10 000	< 29
100 000	100 000	< 36
1 000 000	1 000 000	< 42
10 000 000	10 000 000	< 49
100 000 000	100 000 000	< 56
1 000 000 000	1 000 000 000	< 62

Summary & Take Home Message

Things to remember

- searching in a sorted array
- linear search
- linear search with linear run time in the worst and average case
- binary search
- binary search with logarithmic time in the worst case

Take Home Message

- It pays not to be very stupid.
- It pays to exploit structural properties.
- Binary search is very, very, very much faster than linear search unless we search in very small arrays.

Lecture feedback `http://onlineted.com`